形式化方法的逆袭——如何找出Timsort算法和玉兔月球车中的Bug？

2015-03-30 宾狗 工程 形式化

0x00 背景

形式化方法（Formal Methods）在我们一般人眼中是非常高大上的东西，最多也就是当年在课堂上听说过而已，在实际工作中很少有人使用。

前一阵子Reddit上的一个帖子让高冷的形式化方法也上了一次头条，故事就是国外某技术团队利用形式化方法验证了Java中一些排序算法的正确性，但是在验证Timsort排序算法时发现了Bug，于是便向Java开源社区报告了这个Bug，同时给出了经过形式化方法验证过的修复方案。本来是个皆大欢喜的结局，结果Java开源社区并没采纳他们的修复方案，而是采用了另一个Dirty Solution……（就不听你的，不服你来咬我啊）

回归到这个Bug本身，我们来看看形式化方法是如何大显身手的

0x01 什么是Timsort

说起排序，我们比较熟悉的有冒泡、选择、插入排序，当然还有神奇的快排（Quick Sort），Timsort是个什么鬼？

Timsort算法是Tim Peters于2002年提出的一个排序算法（以自己名字命名的……），相比其他排序算法算是后起之秀了。我们评价一个排序算法的好坏要从许多方面衡量，看看下面这张图

Name	Best	Average	Worst	Memory	Stable
Quicksort	$n \log n$	$n \log n$	n^2	$\log n$	Depends
Merge sort	$n \log n$	$n \log n$	$n \log n$	Depends	Yes
In-place Merge sort	—	—	$n (\log n)^2$	1	Yes
Heapsort	$n \log n$	$n \log n$	$n \log n$	1	No
Insertion sort	n	n^2	n^2	1	Yes
Introsort	$n \log n$	$n \log n$	$n \log n$	$\log n$	No
Selection sort	n^2	n^2	n^2	1	Depends
Timsort	n	$n \log n$	$n \log n$	n	Yes
Shell sort	n	$n(\log n)^2$	$O(n\log^2 n)$	1	No
Bubble sort	n	n^2	n^2	1	Yes
Binary tree sort	n	$n \log n$	$n \log n$	n	Yes
Cycle sort	—	n^2	n^2	1	No
Library sort	—	$n \log n$	n^2	n	Yes
Patience sorting	—	—	$n \log n$	n	No
Smoothsort	n	$n \log n$	$n \log n$	1	No
Strand sort	n	n^2	n^2	n	Yes
Tournament sort	—	$n \log n$	$n \log n$		
Cocktail sort	n	n^2	n^2	1	Yes
Comb sort	—	—	n^2	1	No
Gnome sort	n	n^2	n^2	1	Yes
Bogosort	n	$n \cdot n!$	$n \cdot n! \rightarrow \infty$	1	No

其他乱七八糟的排序算法就不看了（也看不懂……），看看那些眼熟的排序算法，快排虽然平均时间复杂度非常好，但是在最优、最坏时间复杂度以及算法的稳定性上来说都不如Timsort

所以Timsort成为了Python内置的排序算法，后来Java SE 7、Android和GNU Octave都引入了Timsort排序算法。

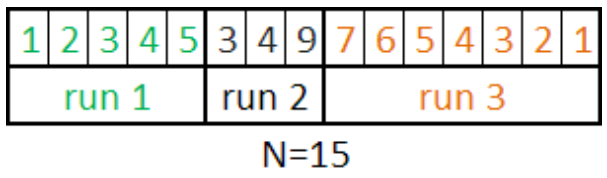
那么这么优秀的一个算法，怎么会出现Bug呢？当然这并非Timsort算法思想的问题，而是写程序的人犯的错误，原因就是Timsort太复杂了（相比其他算法而言），当初写程序的人无意中忽略了一种比较特殊的情况……

要理解这个Bug产生的原因，我们先来看看Timsort算法的原理

0x02 Timsort原理

简单来说，Timsort是一种结合了归并排序和插入排序的混合算法，它基于一个简单的事实，实际中大部分数据都是**部分有序**（升序或降序）的。

Timsort排序算法中定义数组中的有序片段为run，每个run都要求单调递增或严格单调递减（保证算法的稳定性），如下图所示



抛开一些细节，Timsort排序算法可以概括成两步：

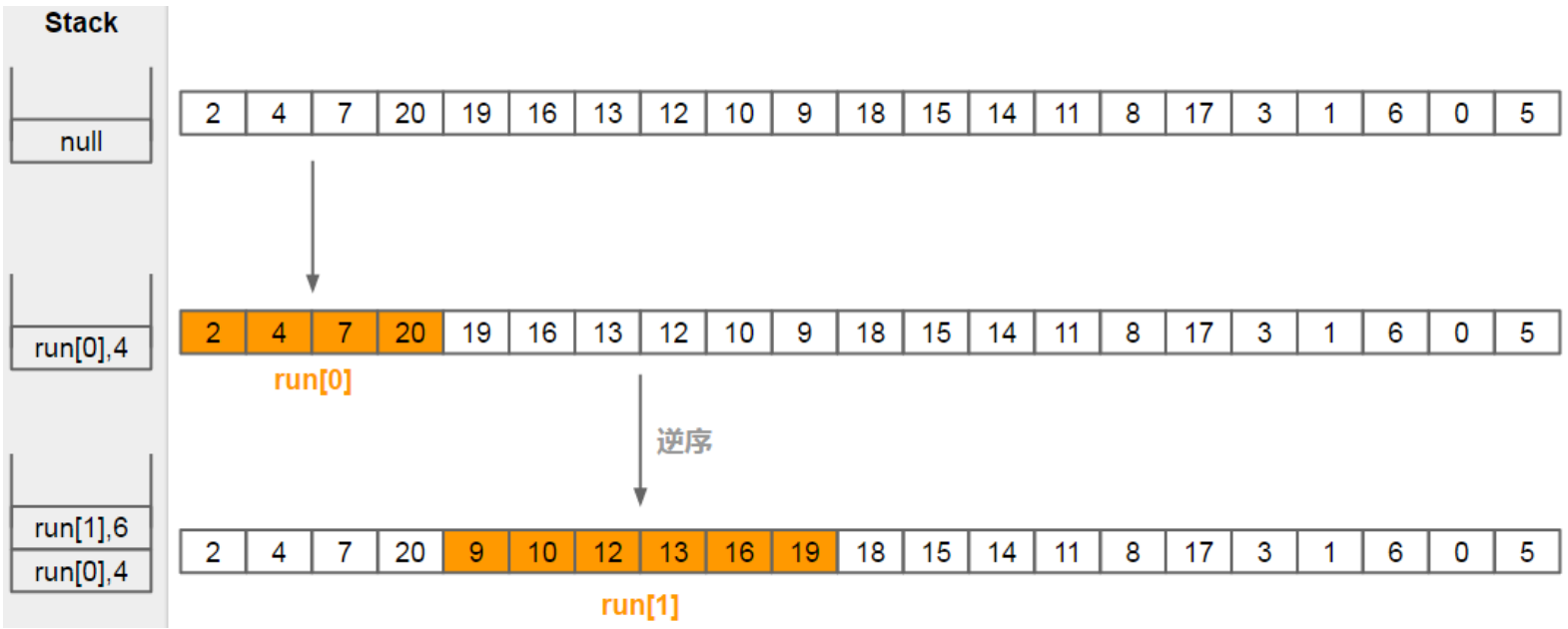
第一步就是把待排数组划分成一个个run，当然run不能太短，如果长度小于minrun这个阈值，则用**插入排序**进行扩充。

第二步将run入栈，当栈顶的run的长度**不满足**下列约束条件中任意一个时，

1. runLen[n-2] > runLen[n-1] + runLen[n]
2. runLen[n-1] > runLen[n]

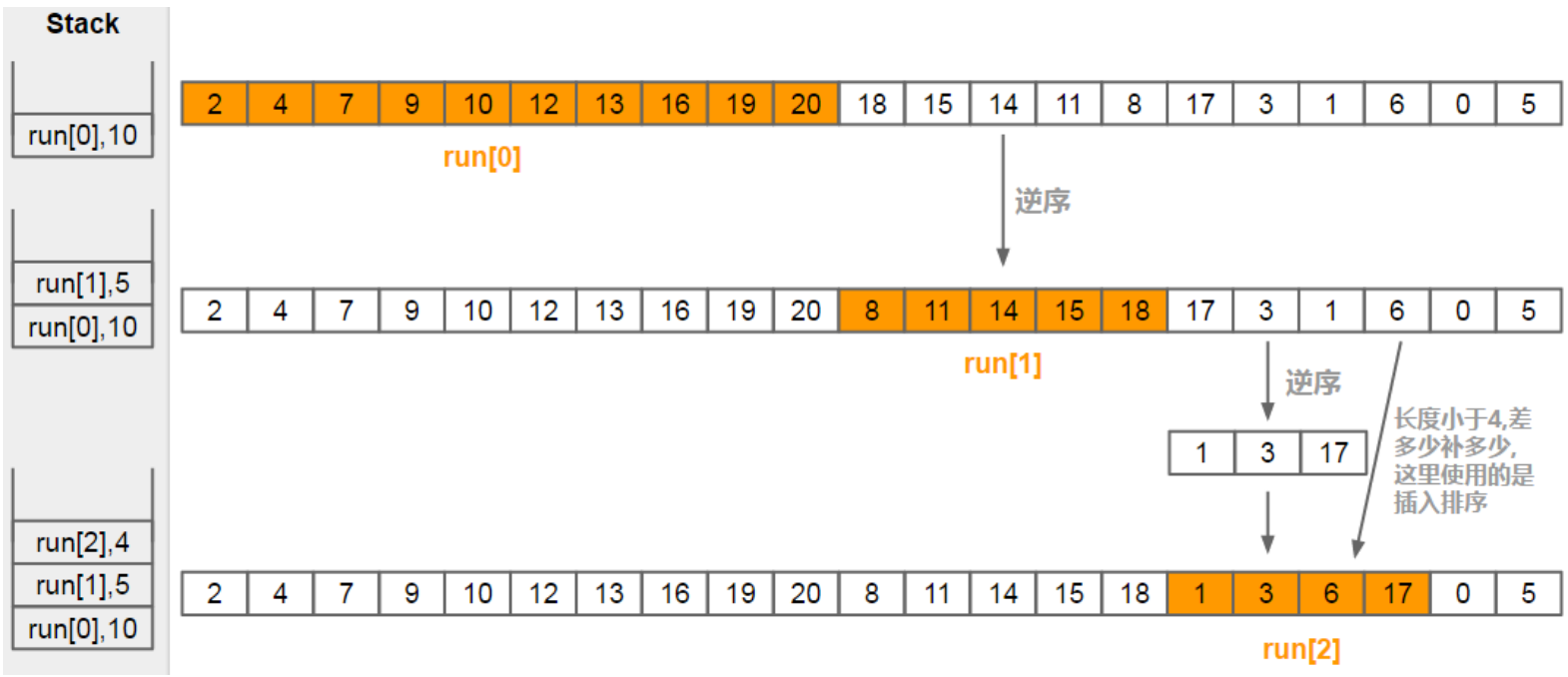
则利用**归并排序**将其中最短的2个run合并成一个新run，最终栈空的时候排序也就完成了。

下面以一个实例进行说明，这个例子中我们设置minrun=4，也就是说run的最小长度不能小于4。每划分出一个run就将其入栈，如下图所示



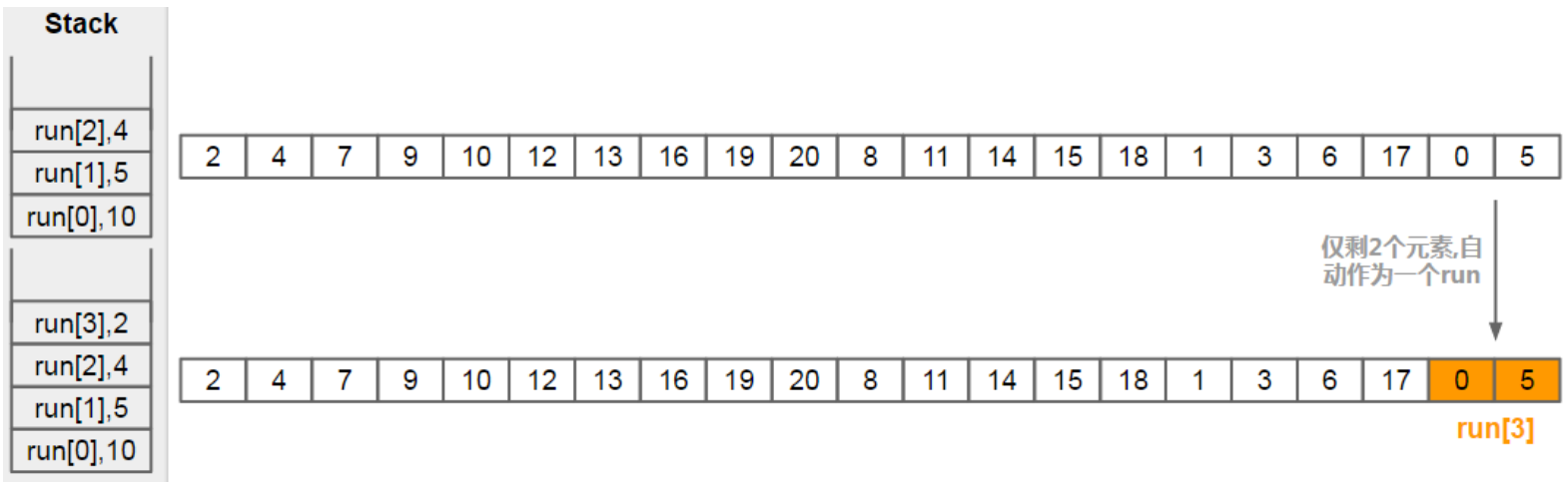
注意，此时栈顶的run是不满足约束条件的，因为此时runLen[0] < runLen[1]，所以要对这两个run进行归并，当然这个过程使用的是归并排序。

如果遇到有序片段长度小于minrun，则要进行补齐，如下图所示

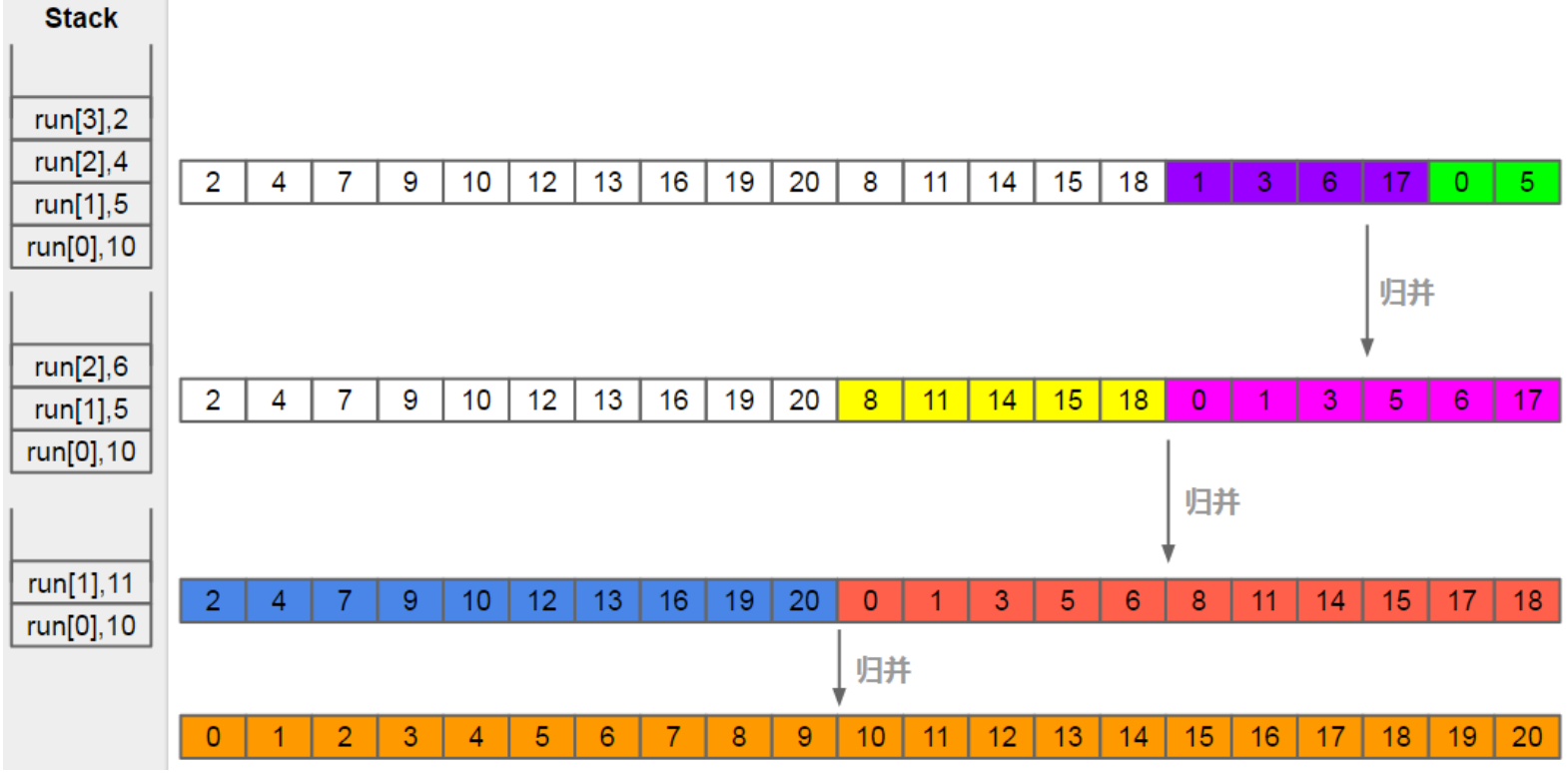


注意，此时栈顶run是满足约束条件的，10 > 5 + 4，5 > 4，因此不需要进行归并。

最后数组元素个数不足minrun了，只能作为一个run了



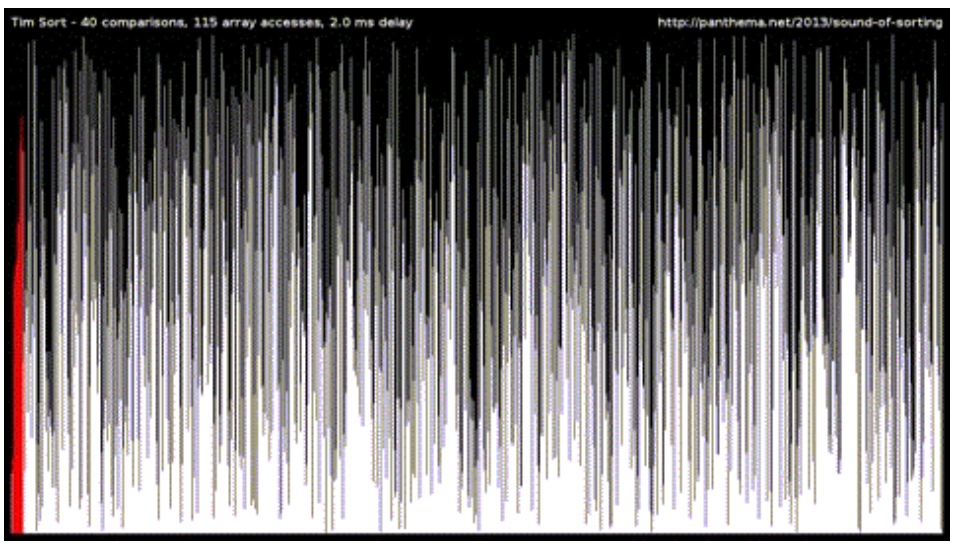
此时栈顶的run又不满足约束条件了，5 < 4 + 2，所以需要进行归并。后续过程如下图所示



这样排序过程就完成了~有的同学可能会有疑问，为什么要有那个奇怪的约束条件呢？每次入栈的时候直接进行归并不行吗？这主要是考虑到归并排序的效率问题，因为将一个长序列和一个短序列进行归并排序从效率和代价的角度来看是不划算的，而两个长度均衡的序列进行归并排序时才是比较合理的也比较高效的。

当然这里我们忽略了许多细节问题，如minrun的长度计算，插入排序和归并排序具体实现中的一些技巧。更多关于Timsort排序算法的细节请参考OpenJDK 源代码阅读之 TimSort

为了更直观的理解Timsor看法的过程，可以看看下面这张git图，是不是与我们刚才描述的算法过程基本一致呢？



还有一个台湾高三学生讲解Timsort算法的视频，里面介绍了不少我们忽略的细节问题

0x03 Timsort中的Bug

了解了Timsort算法的过程和原理，好像没有什么逻辑上的问题，那么这个Bug到底出在哪呢？

这个Bug恰恰出现在那个约束条件上，因为Timsort算法设置这个约束条件的是为了**保证归并排序时两个子序列长度是均衡的**，隐含的一层意思是栈内所有run都应该满足该约束条件（即使不在栈顶），即对2 <= i <= StackSize-1，也必须满足

1. runLen[i-2] > runLen[i-1] + runLen[i]

2. runLen[i-1] > runLen[i]

JDK源码中的注释也说明了这一点

```
/**
 * Examines the stack of runs waiting to be merged and merges adjacent runs
 * until the stack invariants are reestablished:
 *
 *     1. runLen[i - 3] > runLen[i - 2] + runLen[i - 1]
 *     2. runLen[i - 2] > runLen[i - 1]
 *
 * This method is called each time a new run is pushed onto the stack,
 * so the invariants are guaranteed to hold for i < stackSize upon
 * entry to the method.
 */
private void mergeCollapse() {
  while (stackSize > 1) {
    int n = stackSize - 2;
    if (n > 0 && runLen[n-1] <= runLen[n] + runLen[n+1]) {
      if (runLen[n - 1] < runLen[n + 1])
         n--;
      mergeAt(n);
    } else if (runLen[n] <= runLen[n + 1]) {
      mergeAt(n);
    } else {
      break; // Invariant is established
    }
  }
}
```

在大多数情况下，仅检查栈顶的3个run是否满足这个约束条件是可以保证整个栈内所有run均满足该约束条件。但是在一些特殊的情况下就不行了，如下面这个栈（右侧为栈顶）

```
120, 80, 25, 20, 30
```

对照上面的源码，因为25 < 20 + 30，25 < 30，所以将25和20两个run进行合并，此时栈内的情况变为

```
120, 80, 45, 30
```

由于80 > 45 + 30，45 > 30，满足约束条件，此时归并就终止了。但是注意栈里的其他run，120 < 80 + 45，这是不满足约束条件的，而由于我们只判断了栈顶的run，因此在这里就留下了“隐患”。

在大多数情况下，这个问题没什么大不了，不影响我们平时一般的排序操作，因为我们的数据没有那么多，更不会大量出现上述情况。但是这个国外技术团队精心构造了一个Array，成功的让Timsort算法报了java.lang.ArrayIndexOutOfBoundsException这个错误。他们还把复现这个错误的代码放在了github上，代码请戳这里

为什么是这个奇怪的错误？这就和Timsort算法的Java实现中申请栈空间的大小有关系了，看看原始代码

```
int stackLen = (len <    120  ?  5 :
                len <   1542  ? 10 :
                len < 119151  ? 19 : 40);
runBase = new int[stackLen];
runLen = new int[stackLen];
```

其中len表示输入的Array的长度，stackLen表示申请的栈的大小。那么上面的那些边界条件和奇怪的数字是怎么计算出来的呢？

其实很简单，考虑这样一个问题：怎样构造一个序列使其每个元素均满足Timsort算法的约束条件呢？可以设序列中的元素为E(n)，只要满足E(n) = E(n-1) + E(n-2) + 1即可，是不是和Fibonacci数列非常类似？用程序实现一下

```
# -*- coding: utf-8 -*-

# 栈顶之上加一个辅助空run，栈顶的run长度设置为minrun=16
known = {0: 0, 1: 16}
def fib(n):
```

```
    if n in known:
        return known[n]
    res = fib(n - 1) + fib(n - 2) + 1
    known[n] = res
    return res

ns = [5,10,19,39,40]
for n in ns:
    fibsum = 0
    for i in range(n):
        fibsum += fib(i)
    print "n={0},sum={1}".format(n,fibsum)

# 输出
# n=5,sum=119
# n=10,sum=1541
# n=19,sum=119150
# n=39,sum=1802926565
# n=40,sum=2917196495
# ps:Java中int的最大值为2147483648，1802926565 < 2147483648 < 2917196495
```

那么为什么只选取4个区间呢？为什么选择minrun=16呢？我个人认为是为了省事，反正能保证栈空间够用就行。

但是请注意这是在理想条件下，也就是栈内每个run都必须满足这个约束条件。而我们刚才给出了一个反例，那么在反例出现的情况下，用到栈的大小会比我们预想的要大一些。

国外的这个技术团队在论文中算出了最坏情况下用到栈的大小，并画出了一张表

array size	64	128	160	65536	131072	67108864	1073741824
required stack size	3	4	5	21	23	41	49
runLen.length	5	10	10	19 (24)	40	40	40

第二行表示最坏情况下需要用到栈的大小，第三行表示Timsort算法实际给出的栈大小（见上文JDK源码）。有意思的是在Array长度为65536时，最初Java中Timsort设定栈的长度为19，但是后来有人报告了Bug，也就是说这个Bug在实际中是出现过的。然而Java开源社区的程序员可能无法定位这个Bug的根源，只好从表面解决这个问题，在后来的更新中把栈的大小改成了24……（不过确实也解决了问题）但是隐患依然存在的，可以看到在Array长度为67108864时，最坏情况下用到的栈大小41，而Java中Timsort设定的长度为40。所以只要精心构造一个Array，就能触发这个Bug。然而如本文开头所说，Java开源社区的程序员依然不从根本上解决问题，还是用老办法，增加栈的大小……（老子就会这一个技能，不服你来咬我啊）

0x04 如何发现这个Bug

这个Bug是非常隐晦的，除非脑洞大开，否则很少人能从代码上来看出问题。那么测试呢？好像也不太可行，依靠自动化生成的测试集似乎很难生成一个能够触发这个Bug的Array（尤其是在Array长度较大的情况下），这个团队也是在深入研究了这个Bug的基础上才构造出一个能触发Bug的Array。这又回到了一个老问题上，怎样证明一个程序的正确性？程序运行1万次、100万次不出错并不能说明程序没有问题（请参考各类漏洞），因为测试集不能保证覆盖所有执行路径，要证明一个程序的正确性我们必须要采用其他手段。

这就需要用到形式化方法了，事实上有关程序正确性证明的研究早在图灵和冯·诺依曼时期就开始了，当然要真正掌握这个理论需要太多逻辑和演算的知识（什么程序规约、前置断言、后置断言，反正我也不懂）……我们就简单看看这个国外技术团队是怎样用形式化工具KeY来验证Timsort算法的

KeY是为Java平台设计的程序正确性证明工具，使用Java Modeling Language(JML)在程序中加入一些断言，就像这样

```
/*@ private normal_behavior
  @ requires
  @   n >= MIN_MERGE;
  @ ensures
  @   \result >= MIN_MERGE/2;
  @*/

private static int /*@ pure @*/ minRunLength(int n) {
  assert n >= 0;
  int r = 0;      // Becomes 1 if any 1 bits are shifted off
  /*@ loop_invariant n >= MIN_MERGE/2 && r >=0 && r<=1;
    @ decreases n;
```

```
      @ assignable \nothing;
      @*/
    while (n >= MIN_MERGE) {
      r |= (n & 1);
      n >>= 1;
    }
    return n + r;
  }
```

那些前面有一个奇怪的@符号的就是断言了，其实无非就是定义输入要满足哪些条件，输出要满足哪些什么条件。比如对于一个排序算法而言，输入的断言就是一个非空的Array，输出的断言应该是一个有序的Array且元素集合与输入一致。

有人说这样就能保证程序的正确性吗？KeY为什么能覆盖所有的执行路径而软件测试不能？因为KeY并没有生成实际的测试数据集，而是把程序符号化，通过符号逻辑和演算就可以考虑所有可能的执行路径。（此处省略一万字……）

当然即便是使用KeY进行程序正确性证明工作量也不小，原始的Timsort.java文件大约900多行，而加入了JML之后大约1400多行……几乎每个函数、循环、条件语句前面都要加入大量断言。当然这也并非无用功，至少我们发现了这个反程序猿的Bug

当然他们还给出了一个经过KeY验证过的正确的解决方案，业界良心啊，感受一下

```
/*@ private normal_behavior
@ requires
@      \dl_elemInv(runLen, (\bigint)stackSize-4, MIN_MERGE/2)
@   && \dl_elemBiggerThanNext(runLen, (\bigint)stackSize-3)
@   && stackSize > 0;
@ ensures
@    (\forall \bigint i; 0<=i && i<(\bigint)stackSize-2;
@        \dl_elemInv(runLen, i, MIN_MERGE/2))
@ && \dl_elemBiggerThanNext(runLen, (\bigint)stackSize-2)
@ && (   (\sum int i; 0<=i && i<stackSize; (\bigint)runLen[i])
@     == \old((\sum int i; 0<=i && i<stackSize; (\bigint)runLen[i])))
@ && (runLen[(\bigint)stackSize-1] >= \old(runLen[(\bigint)stackSize-1]))
@ && (0 < stackSize && stackSize <= \old(stackSize));
@ assignable
@   this.runBase[1 .. (\bigint)this.runBase.length-1],
@   this.runLen[0 .. (\bigint)this.runLen.length-1],
@   this.stackSize,
@   this.tmp,
@   this.tmp[0 .. (\bigint)tmp.length-1],
@   this.minGallop,
@   this.a[0 .. (\bigint)this.a.length-1];
@*/
  private void newMergeCollapse() {
        /*@ loop_invariant
    @    (0 < stackSize && stackSize <= runLen.length)
    @ && (   (\sum int i; 0<=i && i<stackSize; (\bigint)runLen[i])
    @     == \old((\sum int i; 0<=i && i<stackSize; (\bigint)runLen[i])))
    @ && (\forall \bigint i; 0<=i && i<(\bigint)stackSize-4;
    @        \dl_elemInv(runLen, i, MIN_MERGE/2))
    @ && \dl_elemBiggerThanNext(runLen, (\bigint)stackSize-4)
    @ && \dl_elemLargerThanBound(runLen, (\bigint)stackSize-3, MIN_MERGE/2)
    @ && \dl_elemLargerThanBound(runLen, (\bigint)stackSize-2, MIN_MERGE/2)
    @ && \dl_elemLargerThanBound(runLen, (\bigint)stackSize-1, 1)
    @ && (\forall \bigint i; 0<=i && i<(\bigint)stackSize-1;
    @        (\bigint)runBase[i] + runLen[i] == runBase[(\bigint)i+1])
    @ && (runLen[(\bigint)stackSize-1] >= \old(runLen[(\bigint)stackSize-1]))
    @ && (0 < stackSize && stackSize <= \old(stackSize))
    @ && (this.tmp != null && (tmp != \old(tmp) ==> \fresh(tmp)))
@ && (\typeof(tmp) == \type(Object[]) && \dl_inInt(minGallop));
    @ assignable
    @   this.runBase[1 .. (\bigint)this.runBase.length - 1],
    @   this.runLen[0 .. (\bigint)this.runLen.length-1],
    @   this.stackSize,
```

```
        @   this.tmp,
        @   this.tmp[0 .. (\bigint)this.tmp.length-1],
        @   this.minGallop,
        @   this.a[0 .. (\bigint)this.a.length-1];
        @ decreases stackSize;
        @*/
            while (stackSize > 1) {
                    int n = stackSize - 2;
                    if (n > 0 && runLen[n - 1] <= runLen[n] + runLen[n + 1] || n - 1 > 0 && runLen[n
    - 2] <= runLen[n] + runLen[n - 1]) {
                            if (runLen[n - 1] < runLen[n + 1]) n--;
                    } else if (n < 0 || runLen[n] > runLen[n + 1]) {
                            break; // Invariant is established
                    }
                    mergeAt(n);
            }
    }
```

原先只判断了栈顶的3个run，现在我们判断栈顶4个run，经过这样改动之后即可保证栈内所有run均满足约束条件了，而且请放心大胆的使用，因为这是经过形式化方法验证过的~

0x05 玉兔月球车中的Bug

那么形式化方法这么复杂，花这么大力气只发现了一个小Bug，而且别人压根就不买账，投入和回报有点不成正比啊……没错，形式化方法有时候就是这么吃力不讨好，而且平时也没有哪个程序员会用形式化方法来验证自己的程序。那么到底什么人在用形式化方法呢？就目前来看要么是有钱的大公司，如Amazon、Facebook，要么是有钱的国家级工程项目，也就是我们下面要说的玉兔月球车。

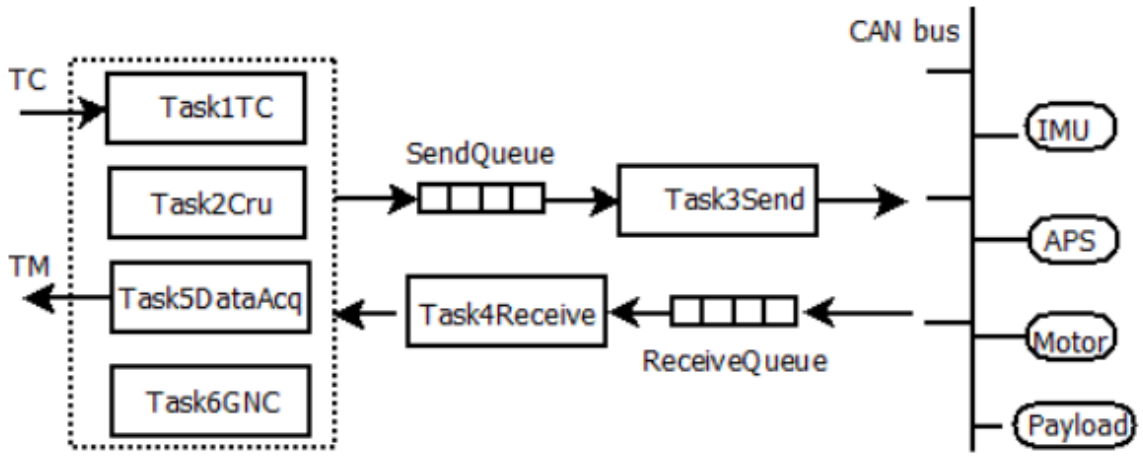
航空航天技术历来都是国家综合实力的一个重要表现，大家都是不计成本的往里砸钱……当然航空航天技术同时也以高风险著称，一个小零件故障也有可能造成严重后果，挑战者号升空后爆炸就是因为一个O型环失效引发的连锁反应，12亿美元的航天飞机瞬间化为乌有，7名机组人员遇难……正因为其风险如此之高，所以哪怕多花点钱能提高1%的安全性也是值得的。

所以形式化方法一个非常重要的市场就是航空航天领域，去年在新加坡举行的第19届形式化方法国际研讨会上，有两个来自中国的团队进行了汇报，巧合的是他们汇报的题目都与玉兔月球车相关，一个与月球软着陆控制相关，另一个与玉兔月球车控制系统相关。而我有幸在其他场合聆听过其中用形式化方法验证玉兔控制系统那个团队的报告，切实感受到形式化方法的强大与复杂。

Morning Tea Break: 10:00-10:30

Session 4A **10:30-12:30, Location: LT 15** **Chair: Sjouke Mauw**	**Session 4B: Industry Track A** **10:30-12:30, Location: SR1 COM1-02-06** **Chair: Peter Gorm Larsen**
Computing Quadratic Invariants with Min- and Max-Policy Iterations:a Practical Comparison *Pierre Roux, Pierre-Loic Garoche*	**Formal Verification of a Descent Guidance Control Program of a Lunar Lander** *Hengjun Zhao, Mengfei Yang, Naijun Zhan, Bin Gu, Liang Zou, Yao Chen*
Precise Predictive Analysis for Discovering Communication Deadlocks in MPI Programs *Subodh Sharma, Vojtech Forejt, Daniel Kroening, Ganesh Narayanaswamy*	**MDP-based Reliability Analysis of an Ambient Assisted Living System** *Yan Liu, Lin Gui, Yang Liu*
Quiescent Consistency: Defining and Verifying Relaxed Linearizability *John Derrick, Heike Wehrheim, Brijesh Dongol, Gerhard Schellhorn, Bogdan Tofan, Oleg Travkin*	**Formal Verification of Lunar Rover Control Software Using UPPAAL** *Lijun Shan*
The VerCors Tool Set for Verification of Concurrent Programs (tool paper) *Stefan Blom, Marieke Huisman*	**Diagnosing Industrial Business Processes: Early Experiences** *Suman Roy, A. S. M. Sajeev, Srivibha Sripathy*

下图是玉兔月球车控制系统的一个简化版本（真实系统中有30多个应用任务），这里只列出6种

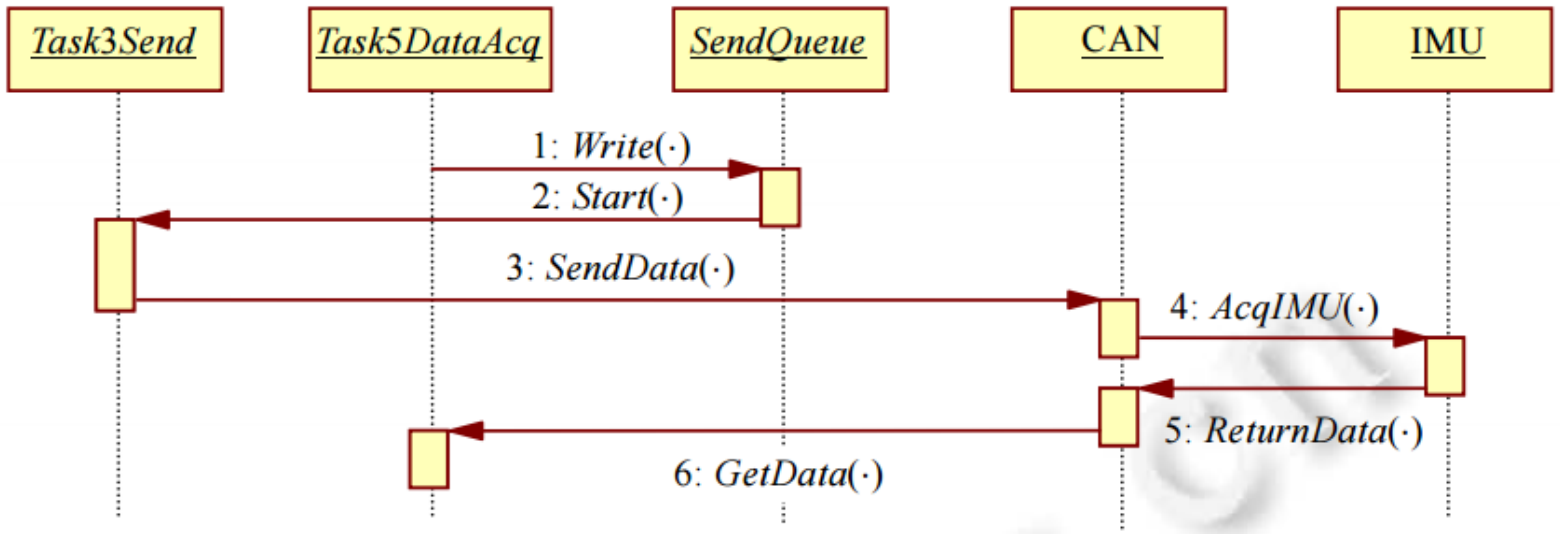


图中左侧主要是一些任务，中间的是消息队列（包括发送队列和接收队列），右侧是CAN bus总线，控制命令和传感器数据都是通过它传递的。具体的含义见下表：

任务名	功能	就绪条件/周期(ms)
Task1TC(遥控任务)	接收从地面不定期发送的 4 帧遥控指令,写入 *SendQueue*	有遥控指令
Task2Cru(重要数据管理任务)	将 8 帧重要数据写入 *SendQueue*	5 000
Task3Send(消息发送任务)	将数据队列 *SendQ* 中的全部数据发送到 CAN 总线的端口	数据到达 *SendQueue*
Task4Receive(消息接收任务)	接收 CAN 总线返回的数据并存储到 *ReceiveQueue* 中	CAN 端口有返回数据
Task5DataAcq(请求数据任务)	将“向敏感器请求数据”消息(1 帧)写入 *SendQueue*	200
Task6GNC(导航任务)	制导、巡航与控制	200

注意从Task1到Task6的优先级是依次递减的，Task1的优先级为6，Task6的优先级为1。

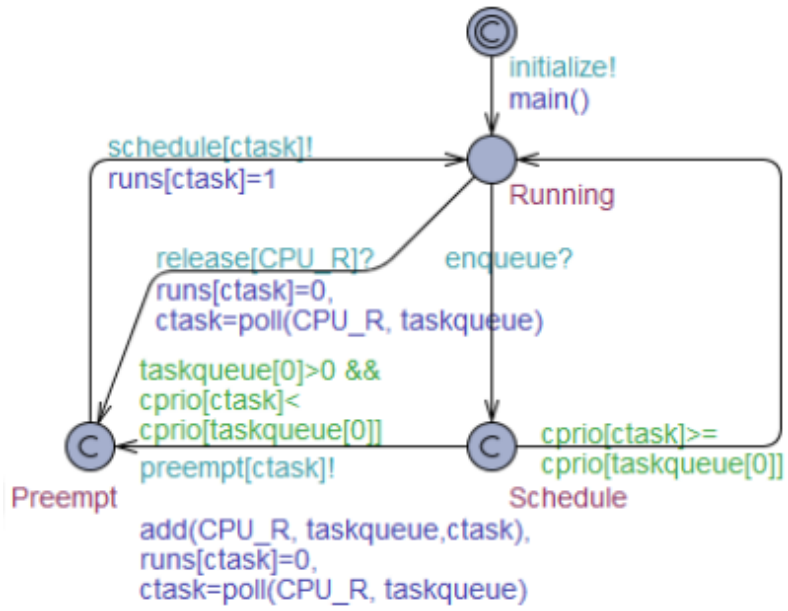
Task5是一个周期性请求数据的任务，预期能4ms接收到完整的遥测数据，如下图所示

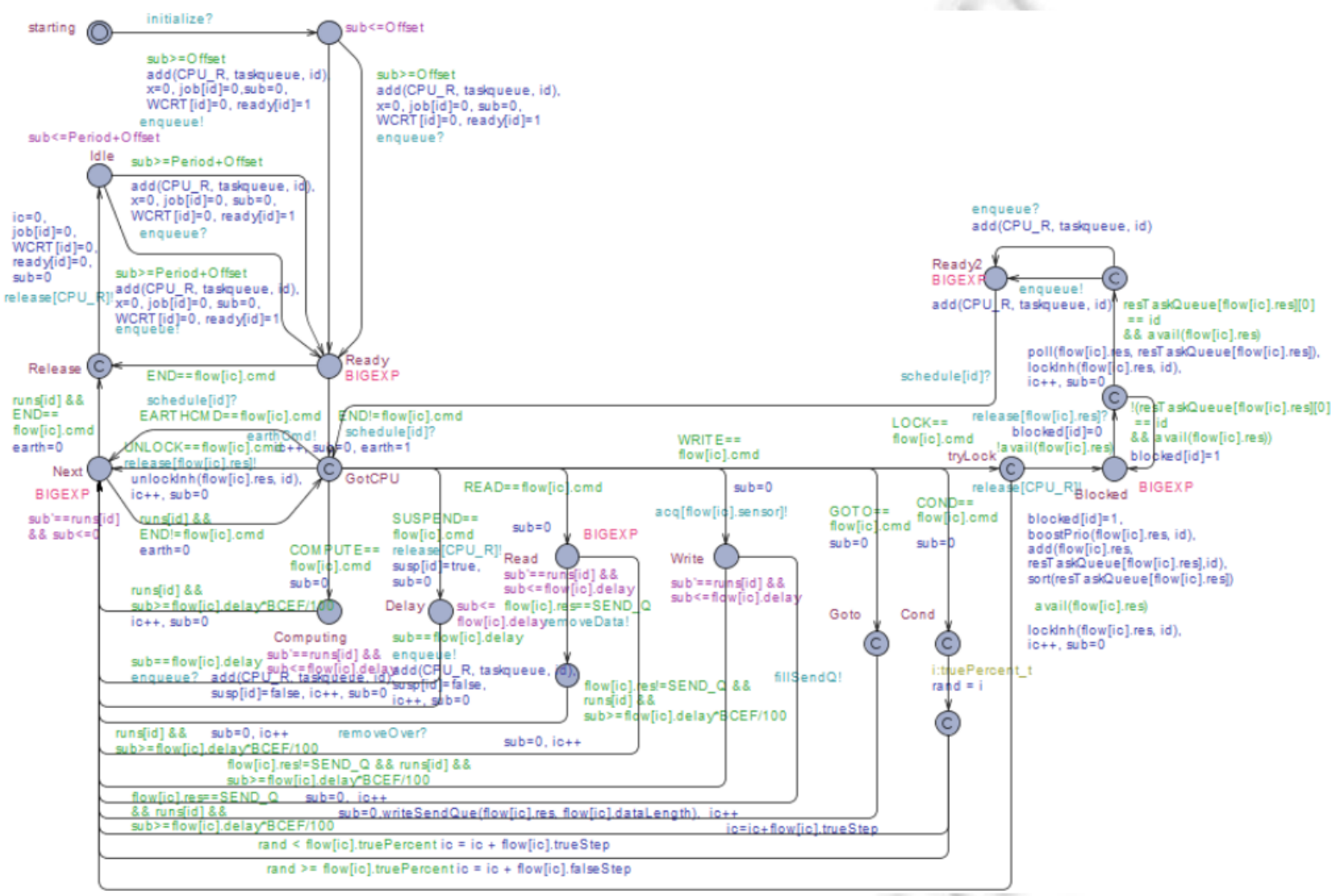


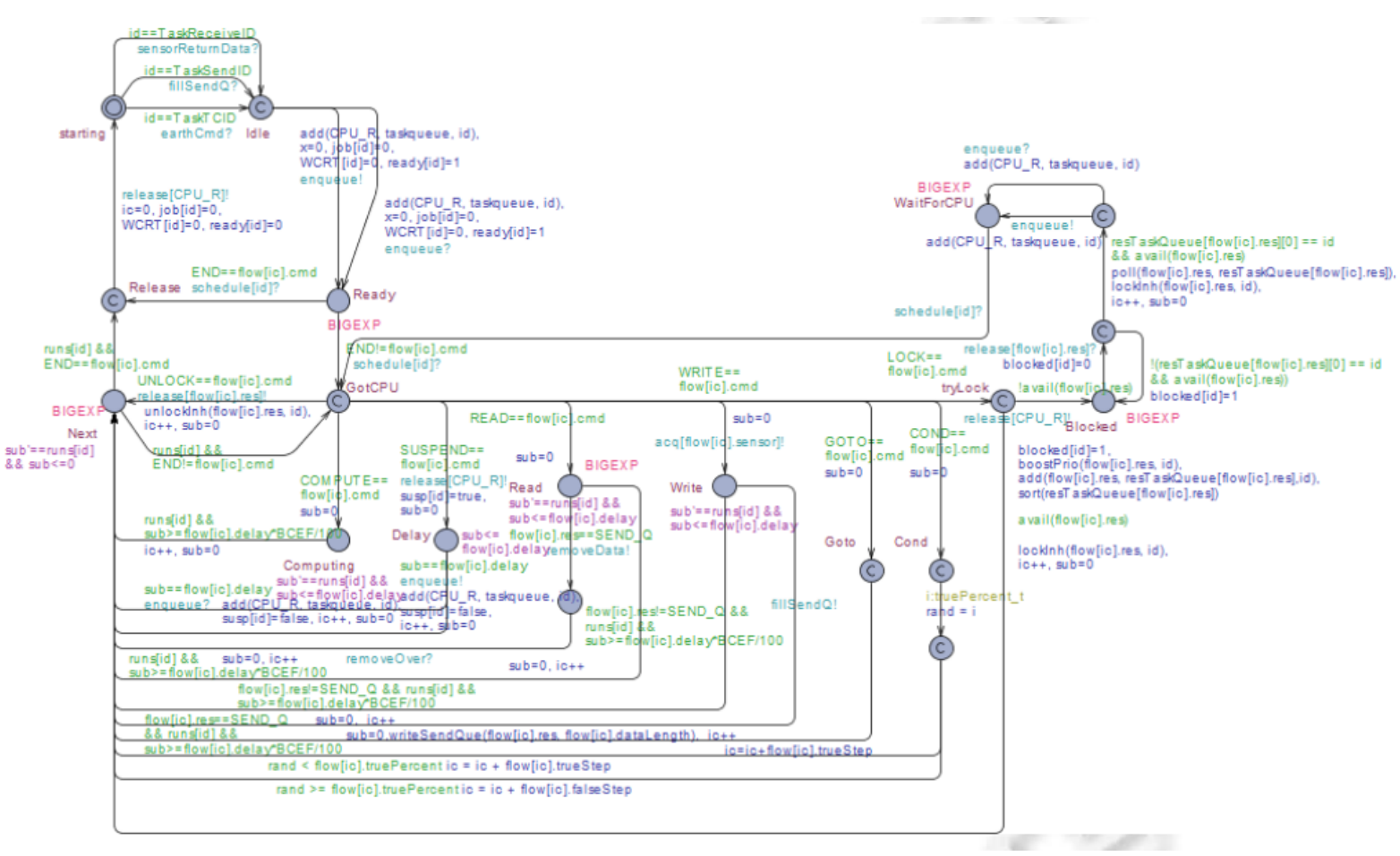
在正常情况下，发送数据1帧，接收数据6帧。这一过程耗时1×0.192(Task3Send)+0.5(IMU响应时间)+6×0.192(Task5)=1.844ms，预设的4ms等待时间是足够的。但是在几年的开发和测试过程中，开发人员偶然观察到几次“遥测超时错误”：即预计能够在4ms内完成的Task5发生超时，未能获取到完整数据。

由于这个系统十分复杂，且操作系统的任务调度具有随机性，多任务系统的某次执行难以重现，要观察实时操作系统对任务的调度非常困难。更重要的是，这个错误出现的次数很少，一年可能才出现一次，很难总结错误出现的条件。

然后又是形式化方法出场了，大家请看下面几张图







关于这些图不想说太多（此处省略一万字……），有人说这不是自动机嘛，没错，自动机本来就是形式化方法中非常重要的一部分。第一个图是抢占式任务的时间自动机模型，对应Task1，第二个图是周期性任务的时间自动机模型，对应是Task2、Task5和Task6，第三个图是偶发任务的时间自动机模型，对应Task3、Task4

据该团队的女博士介绍，画这几张图花了她半年的时间，而且还是在有工具辅助的情况下！

当然最终问题的原因找到了，在T时刻，满足以下条件时，出现遥测错误：

1. Task6在T时刻就绪，Task2和Task5在8ms后就绪
2. T+7ms后，有地面命令（Task1）

在这种情况下，由于Task1和Task2优先级均高于Task5，分别向SendQueue写入4帧和8帧数据，此后Task5才被调度，向SendQueue写入1帧数据，此时SendQueue中已经累积了13帧数据，耗时13×0.192+0.5+6×0.192=4.148ms，超过了4ms，因此发生了超时错误。解决方法也很简单，提高Task3的优先级，让其及时将数据发送出去即可避免此类错误。

别看我们说起来轻描淡写的，这个问题困扰了工程师们好几年，用形式化方法验证也花了半年多的时间，而且只是对其中6个进程进行了形式化验证！如果要对全部30多个进程进行正确性验证，那画面太美我不敢看……

0x06 总结

其实写到这里发现有点跑题，本文并没有讲太多关于形式化方法本身的东西……从我个人的角度来看，咱们一般人确实也没有必要花大力气去研究它，但是却有了解它的必要，当你遇到诡异却又无法解决的问题而绝望时，不妨找找研究形式化方法的专业人士，或许会有转机。

就目前而言，形式化方法的门槛过高，很难将其大规模应用。但可以想像，如果有一天，形式化方法能够在漏洞挖掘、互联网安全等领域普及，必将引发席卷计算机安全领域的风暴。

0x07 参考资料

1. Proving that Android’s, Java’s and Python’s sorting algorithm is broken (and showing how to fix it)
2. OpenJDK’s java.utils.Collection.sort() is broken: The good, the bad and the worst case
3. Formal Verification of Lunar Rover Control Software Using UPPAAL
4. 信息物理融合系统控制软件的统计模型检验
5. OpenJDK 源代码阅读之 TimSort